Design Document of the Trajectory Module in CLARAty

Mihail Pivtoraiko

January 14, 2006

1 Definitions

One of many features of CLARAty is the support for a variety of devices used in robotics, including passive and actively controlled devices. An abstract *controlled device* is defined to be any CLARAty device that contains one or more parameters whose values must be varied with time in a systematic fashion. The manner in which such parameters are varied with time is defined by the control scheme of the device. Control of any device involves specifying a description of the parameter variation, usually through a closed-form mathematical expression of parameter value as a function of time. Such an expression is known as a *trajectory*. Parameter variation could also be accomplished through specifying the variation of any derivative of the parameter, and in this case the trajectory is used to represent this derivative.

The problem of determining the mathematical expression that specifies a trajectory is referred to as *trajectory generation*. Given the desired value of the controlled parameter and the constraints of the system, trajectory generation determines the complete variation of parameter value over time (system constraints largely define this problem, since with no constraints, attaining the desired parameter value would be trivial).

A related problem of affecting the parameter variation on the actual system is referred to as *trajectory following*, also known as servoing.

CLARAty is designed to support any controlled device, and hence its interface components are as general as possible. CLARAty provides a module, named Trajectory, to accomplish both goals of software trajectory generation and trajectory following. This document outlines the design of this module, with the emphasis on its capabilities to support arbitrary controlled devices and to allow this module to be extended and specialized to implement any desirable type of trajectories. In terms of C++ programming, this module is an abstract base class that implements basic framework for trajectory generation and following, such that it can be derived by specialized trajectory implementations and used by any controlled device. In this manner, the goal of the Trajectory module is to abstract the details of system integration, and simplify the development of the trajectory and device software in the spirit of modularization.

2 Requirements

The requirements for the Trajectory module are as follows:

1. Trajectory generation
   (a) In general, system parameters may be coupled, and the trajectory generation must consider all such parameters[1]. The desired parameter values are specified with a vector. In case that only a single parameter is to be controlled, the vector is unit length.
   (b) It may also be necessary to control a time derivative of a parameter. In this case, the time derivative is considered as a controlled parameter itself.
   (c) System constraints are specified as limits on the parameter values and any of its derivatives

[1]Examples of such trajectory generation problems include manipulator placement in Cartesian coordinates and the motion of a car-like vehicle. In these cases, the desired position depends on all controlled parameters at the same time: joint and wheel positions, respectively.

   (d) For trajectories of two or more coupled parameters, the relationship between the parameters must be specified by system Jacobians and Hamiltonians, in the form of matrices.

2. Trajectory following
   (a) Most modern devices are controlled using sampling-based methods: a controller is driven through a series of position values, and updates are frequent enough such that the time-variation is smooth. Trajectory module supports such devices by providing the value of the controlled parameter given any value of time with respect to the start of the trajectory.
   (b) Random time access is supported for obtaining the corresponding parameter values. Although typical control applications will vary the time input monotonically, this is not a requirement. In particular, it is possible to use the same trajectory to return tothe original parameter value.

3. Multi-setpoint trajectories
   (a) It is allowed to specify a series of trajectory generator inputs: desired parameter values and system constraints. Optionally, a desired time to achieve each parameter value can be specified. If timing or other constraints cannot be satisfied, an error will be returned.
   (b) Trajectory module will call upon its specialized trajectory generator in order to generate each of the trajectories in the series and manage the storage of the results.
   (c) Once the sequence has been generated, it is possible to query the parameter value at any time along the trajectory.

4. Trajectory Interruption
   (a) Trajectory following can be interrupted by specifying another trajectory.
   (b) The cascade of interrupted trajectories is treated as a single trajectory. However, it is impossible to move back in time and request the parameter value of a previous trajectory that has already been interrupted. In order to support random time access of a trajectory with multiple set-points, a multi-setpoint trajectory must be used.
   (c) It is the user's responsibility to guarantee that the generation of the new trajectory is sufficiently fast such that trajectory following is not disrupted.
   (d) If the parameter frame of reference of the controlled device is different from that of the Trajectory module[2], then the user is responsible to maintain the coordinate transformation for subsequent interrupting trajectories the same as for the first one[3]. For example, in position control of a wheel, if a trajectory was generated in relative coordinates with respect to some current position of the wheel, q_{offset}, then this offset value must be maintained, since a sequence of interrupted trajectories is considered to be a single trajectory coordinate-wise.

3 Interfacing

The purpose of the Trajectory module is to allow utilization of a variety of trajectory generators provided by CLARAty contributors while at the same time ensuring their compatibility with the rest of CLARAty. This module attains this goal through enforcing the Requirements above and providing a common interface for users of Trajectory and specialized trajectory implementations that use Trajectory as the base class.

3.1 Specialized Trajectory

1. Have access to the information for generation 2. have acccess to time for following 3. Assume starting from origin in configuration space and time. Initial velocity could be non-zero.

[2]For example, when the trajectory is specified in relative coordinates with respect to some value of the parameter

[3]Avoiding doing this will introduce intermixing physical and desired coordinates in the trajectory blending, which will lead to disruptions in trajectory following.

compute_profile()	The arguments are as currently (list). Here we assume 0 position, 0 time, but non-0 derivatives. This is important because all initial time derivatives of position specify the motion (defined as change of position), but initial position itself does not. Therefore, different initial time derivatives must necessarily result in a different motion.
compute_setpoint(time, ...)	Preserve current interface (todo: Explain)

3.2 Device Adaptation of Trajectory

Here we summarize the interfacing API for the users of the Trajectory module. This primarily pertains to motors and other actuators that will setup the Trajectory with the desired initial conditions and then use its output to drive their controlled parameters.

void start(_start_position=0)	This function begins the trajectory. The trajectory generator is invoked first to solve for the trajectory satisfying the goal position and system constraints specified previously. If the user of the Trajectory module is operating in the absolute coordinates and the position of the system is not zero when this function is invoked, then the current position of the system must be specified as an argument. The user also has the option of doing coordinate transformation on his own, thereby assuming that the trajectory output will be with respect to zero. To support this mode conveniently, the _start_position argument is optional, and will assumed to be zero if not specified. This feature also assists in backward-compatibility.
void stop(_deceleration=0)	This function stops the trajectory with a specified deceleration. This parameter is optional, and if not specified, the maximum deceleration allowed by the system, d_{max}, is used. If the argument is specified, then the applied deceleration is $min(\_deceleration, d_{max})$.
void set(position, derivatives)	Preserve current functionality (todo: Explain)
void set_constraints(vector¡Matrix¿)	Preserve current functionality (todo: Explain)
void get_setpoint(&time, ...)	Preserve current functionality (todo: Explain)
void get_setpoint(...)	Preserve current functionality (todo: Explain)
bool is_started()	Preserve current functionality (todo: Explain)
void reset()	Preserve current functionality (todo: Explain)
void advance_timer_by(time)	Preserve current functionality (todo: Explain)